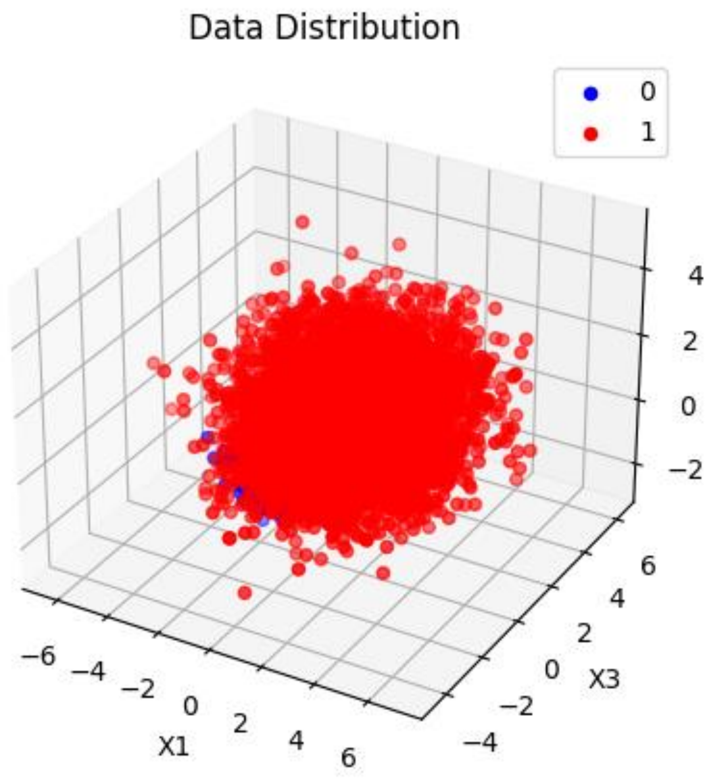
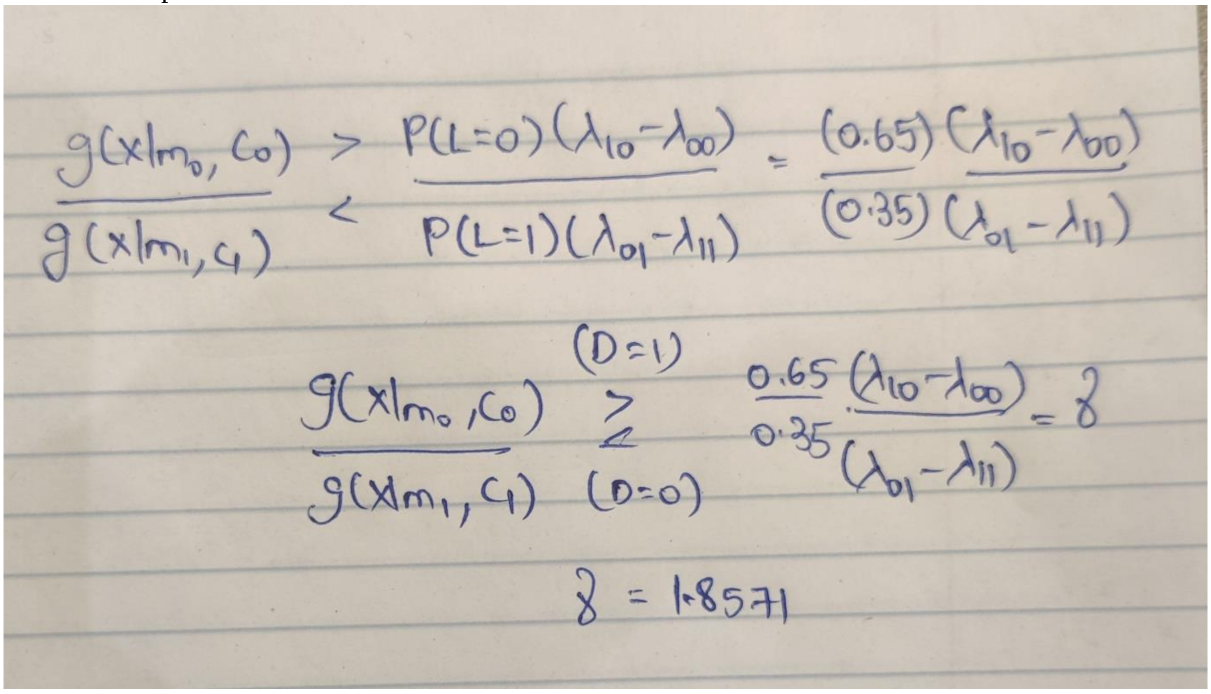
1 (A).



1. Minimum Expected Risk Classification Rule is as follows:

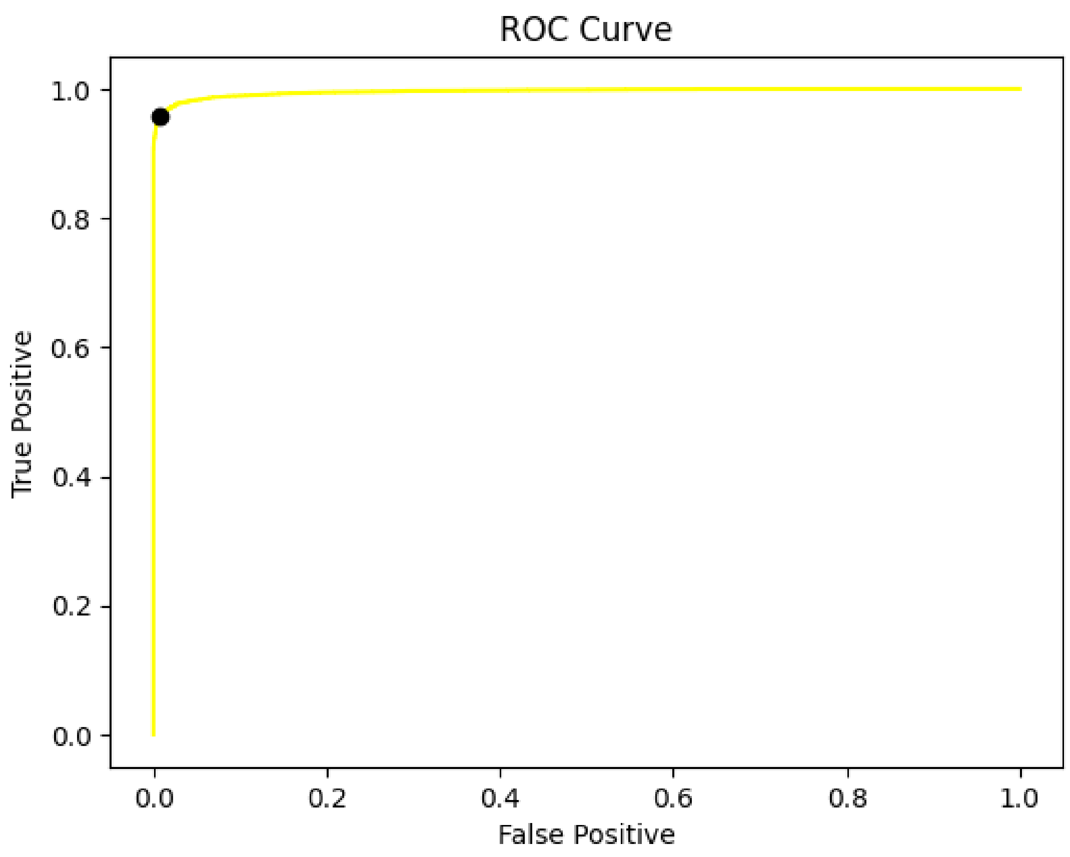


$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \gtrless \frac{P(L=0)(\lambda_{10}-\lambda_{00})}{P(L=1)(\lambda_{01}-\lambda_{11})} = \frac{(0.65)(\lambda_{10}-\lambda_{00})}{(0.35)(\lambda_{01}-\lambda_{11})}$$

$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \underset{(D=0)}{\overset{(D=1)}{\gtrless}} \frac{0.65}{0.35} \frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})} = \gamma$$

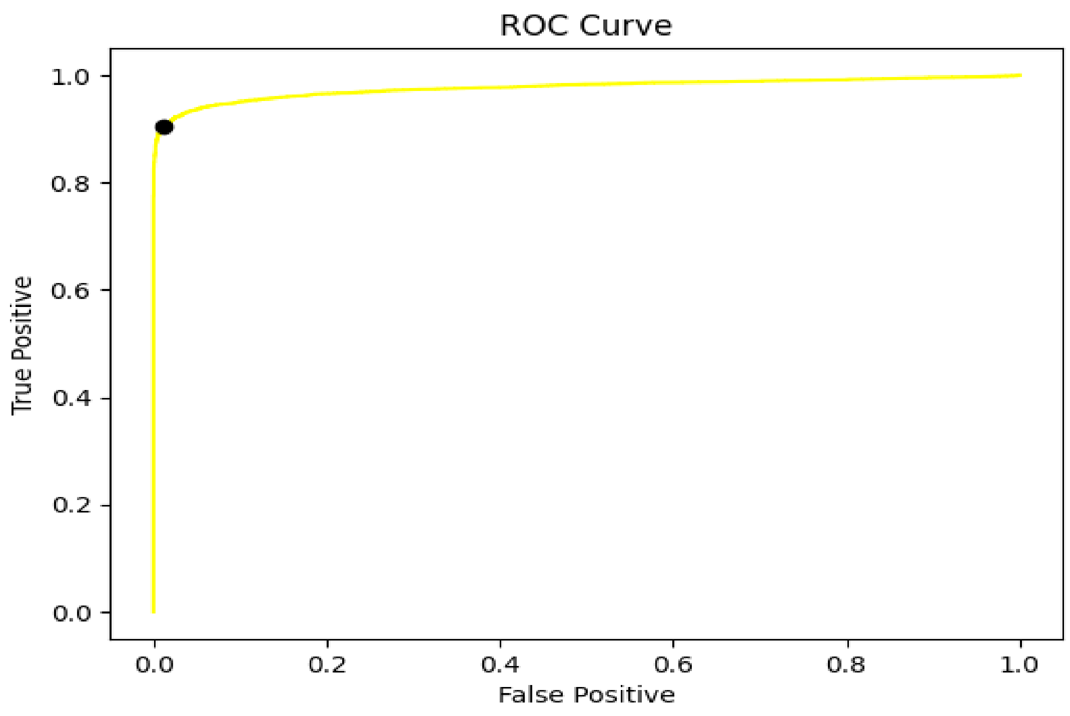
$$\gamma = 1.8571$$

2. The ROC curve is generated by approximating the variation of threshold value γ from 0 to ∞. Practically, γ values are sorted and the mid-points between consecutive two values are considered as threshold points ranging from minimum to maximum.
3. Theoretically, the value of γ is found out by dividing the class priors (0.65/0.35) which is equal to 1.85714. With this threshold, false positives and true positives are computed which help in estimating the Minimum P(error). It can be observed that there is negligible difference in the theoretical and experimental values.



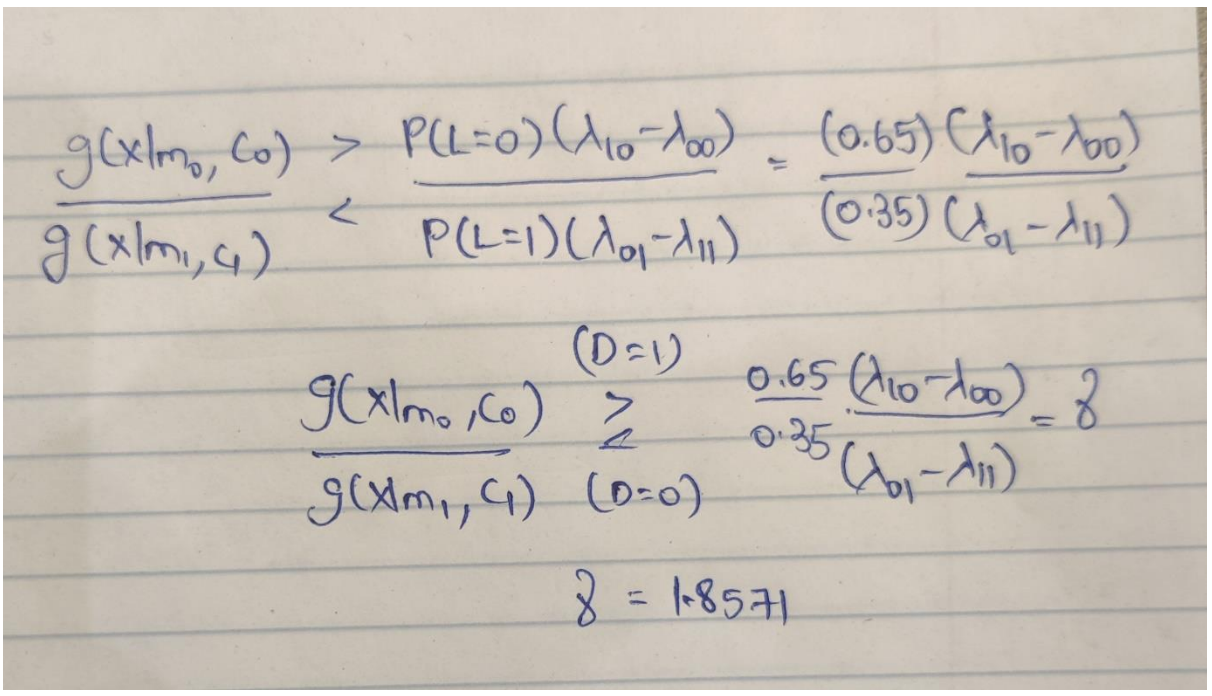
Value of Gamma (practical) is 1.86554
Corresponding minimum error is 0.01961
Value of Gamma (Ideal) is 1.85714
Corresponding minimum error is 0.01961

B.



Value of Gamma (practical) is 0.42374
Corresponding minimum error is 0.04101
Value of Gamma (Ideal) is 1.85714
Corresponding minimum error is 0.05938

1. Minimum Expected Risk Classification Rule is as follows:



$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \gtrless \frac{P(L=0)(\lambda_{10}-\lambda_{00})}{P(L=1)(\lambda_{01}-\lambda_{11})} = \frac{(0.65)(\lambda_{10}-\lambda_{00})}{(0.35)(\lambda_{01}-\lambda_{11})}$$

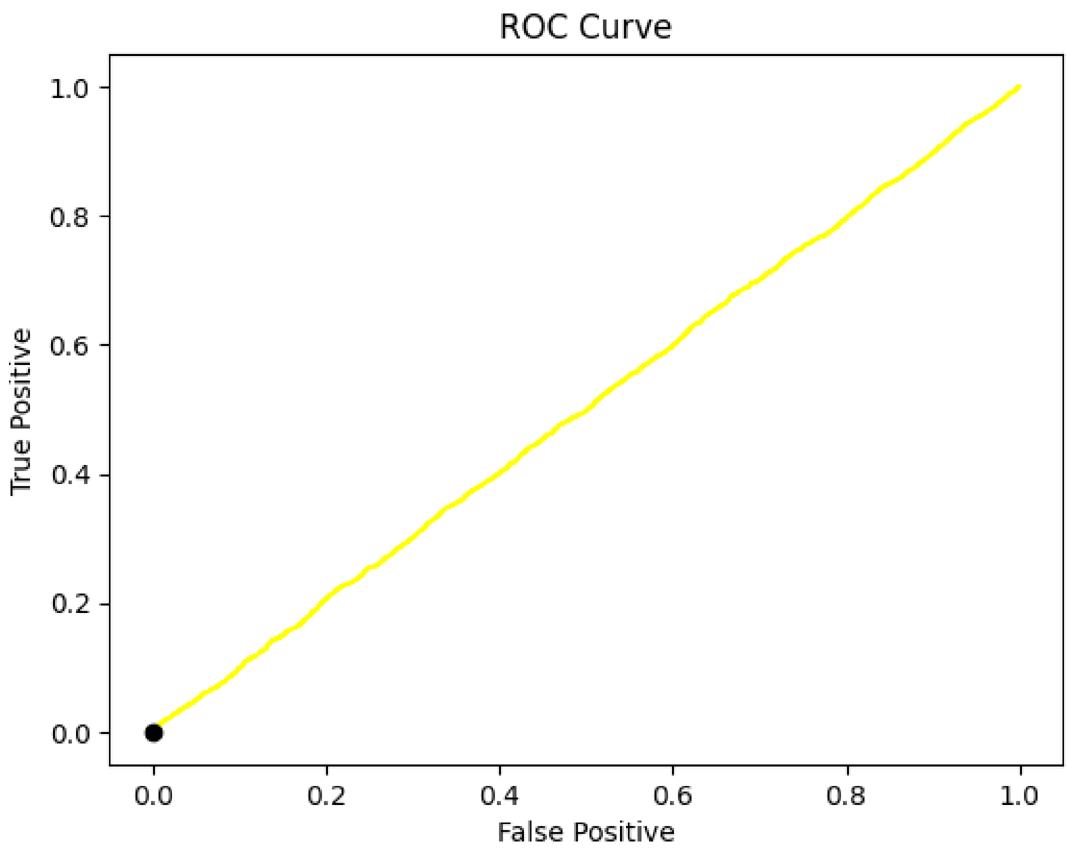
$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \overset{(D=1)}{\underset{(D=0)}{\gtrless}} \frac{0.65}{0.35} \cdot \frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})} = \gamma$$

$$\gamma = 1.8571$$

C.

1. LDA Classification Rule is as follows:

(D = 1) transpose(w_ LDA) X > τ (D = 0)



Value of Tau (practical) is 4.50229
Corresponding minimum error is 0.35018
Value of Tau (Ideal) is 1.85714
Corresponding minimum error is 0.42054

2.

Output: LOSS MATRIX A

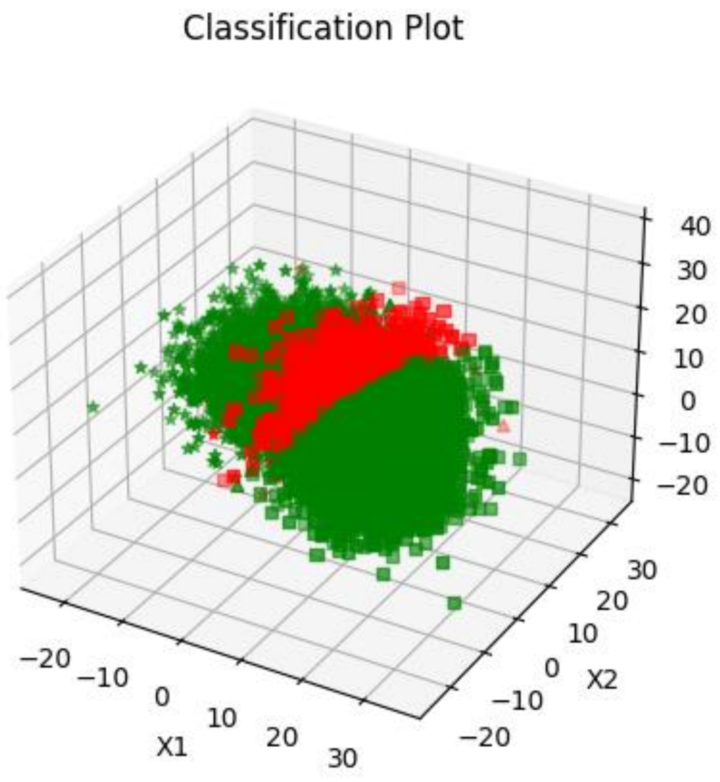
```
1. LOSS MATRIX A :
[[0. 1. 1.]
 [1. 0. 1.]
 [1. 1. 0.]]
2. LOSS MATRIX B :
[[ 0  1 10]
 [ 1  0 10]
 [ 1  1  0]]
3. LOSS MATRIX C :
[[  0   1 100]
 [  1   0 100]
 [  1   1   0]]
```

Select 1 or 2 or 3 : 1

The loss matrix used is :
```
[[0. 1. 1.]
 [1. 0. 1.]
 [1. 1. 0.]]
```

The Average Expected Risk is : 0.08196056747906837

```
Confusion Matrix :
 [[0.93577982 0.03244838 0.26302284]
 [0.02948886 0.92690921 0.020272  ]
 [0.03473132 0.04064241 0.71670516]]
```



Output: LOSS MATRIX B

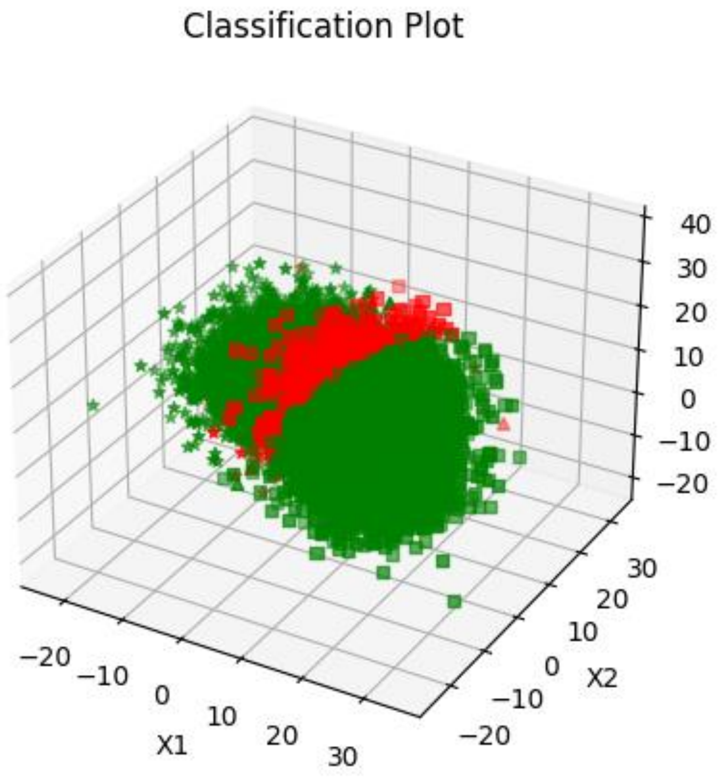
```
1. LOSS MATRIX A :
[[0. 1. 1.]
 [1. 0. 1.]
 [1. 1. 0.]]
2. LOSS MATRIX B :
[[ 0  1 10]
 [ 1  0 10]
 [ 1  1  0]]
3. LOSS MATRIX C :
[[  0   1 100]
 [  1   0 100]
 [  1   1   0]]
```

Select 1 or 2 or 3 : 2

The loss matrix used is :
```
[[ 0  1 10]
 [ 1  0 10]
 [ 1  1  0]]
```

The Average Expected Risk is : 0.1870598242644649

```
Confusion Matrix :
 [[0.85353866 0.02490987 0.13471901]
 [0.02260813 0.83546378 0.00436233]
 [0.12385321 0.13962635 0.86091866]]
```



Output: LOSS MATRIX C

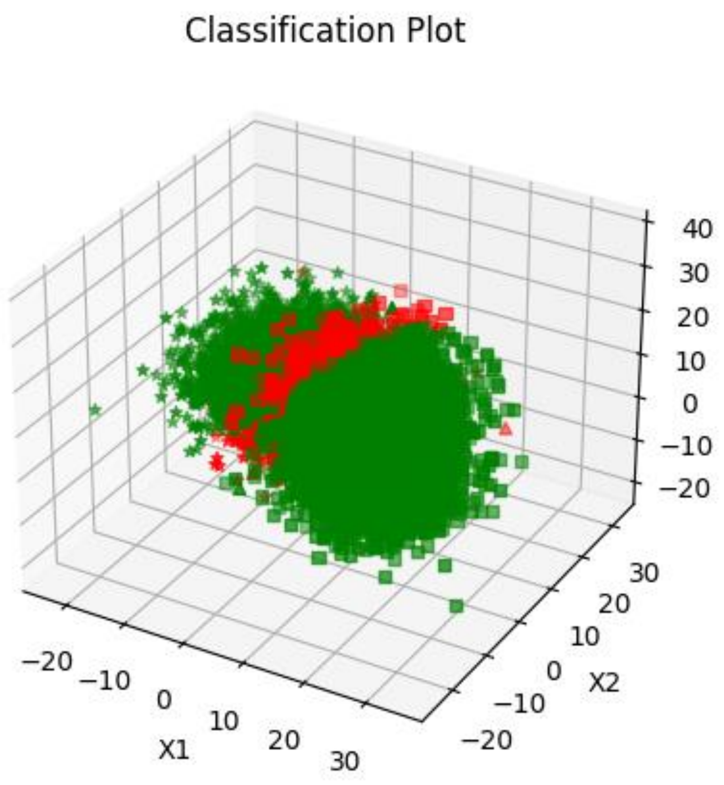
```
1. LOSS MATRIX A :
[[0. 1. 1.]
 [1. 0. 1.]
 [1. 1. 0.]]
2. LOSS MATRIX B :
[[ 0  1 10]
 [ 1  0 10]
 [ 1  1  0]]
3. LOSS MATRIX C :
[[  0   1 100]
 [  1   0 100]
 [  1   1   0]]
```

Select 1 or 2 or 3 : 3

The loss matrix used is :
```
[[  0   1 100]
 [  1   0 100]
 [  1   1   0]]
```

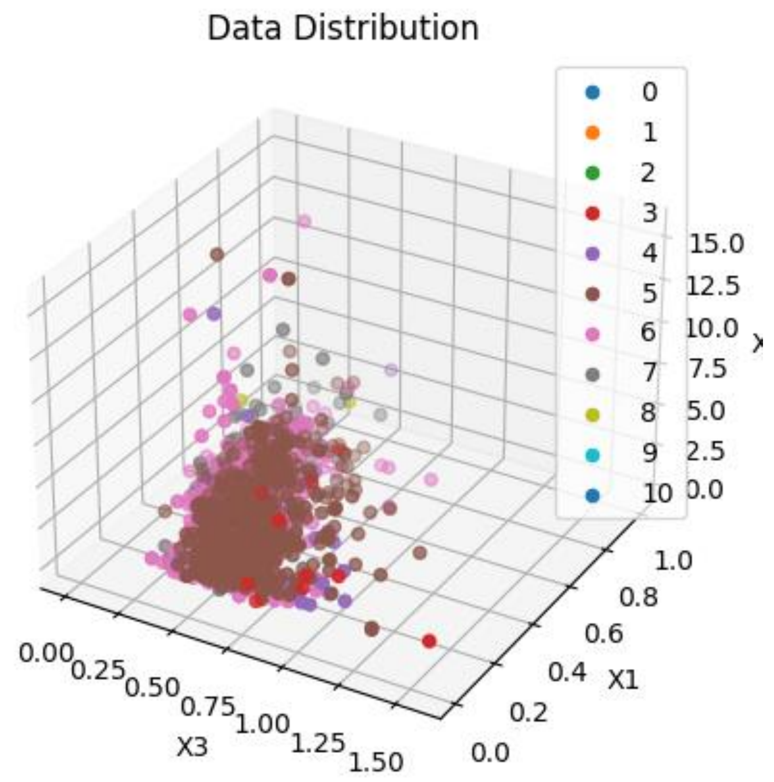
```
The Average Expected Risk is : 0.33782981771787707

Confusion Matrix :
 [[0.67988204 0.01573255 0.05491404]
 [0.01310616 0.6545395  0.        ]
 [0.3070118  0.32972796 0.94508596]]
```



3.

A. Dataset Link - https://archive.ics.uci.edu/ml/datasets/Wine+Quality. The Red Wine Dataset file was used to train the model.



The Average Expected Risk is 0.3240594079151414

Using the sample count, the class priors are computed using the following formula:
P(L) = Numbers of samples belonging to class L / Total Number of samples

On testing the conditionality of the co-variance matrices, it is identified that majority of them yie ld a very large conditional number. Therefore, it is essential to add a small regularization value t o broaden the distribution.

CRegularized = CSampleAverage+λI

In this context, λ represents a hyper-parameter that determines the level of regularization applied to the original variance values. To determine λ, I opted to compute the arithmetic mean of the ma trix's non-zero eigenvalues. The trace, or the sum of the matrix's diagonal elements, is equivalent to the sum of its eigenvalues, while the rank is the number of non-zero eigenvalues. Hence,

Arithmetic Average = trace (CSampleAverage) / rank (CSampleAverage)
λ = α (Arithmetic Average)

The paragraph describes a hyperparameter, represented by the symbol α, which is a real number between 0 and 1. This hyperparameter controls another hyperparameter, λ, which can be adjuste d within the range of 10-3 to 10-9 to optimize the accuracy of the model. The chosen loss functio n is the 0-1 loss, which assigns an equal penalty to all incorrect decisions and no penalty to corre ct decisions. Based on this, a classifier known as the Maximum A Posteriori (MAP) classifier ha s been designed to address the classification problem. Confusion Matrix is

Confusion Matrix is:

```
[[0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.        ]
 [0.         0.         0.         1.         0.         0.00146843
  0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.16981132 0.02496329
  0.01567398 0.00502513 0.         0.         0.        ]
 [0.         0.         0.         0.         0.41509434 0.56975037
  0.17711599 0.01507538 0.         0.         0.        ]
 [0.         0.         0.         0.         0.39622642 0.38913363
  0.70219436 0.53768844 0.16666667 0.         0.        ]
 [0.         0.         0.         0.         0.01886792 0.01468429
  0.0846395  0.36180905 0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.02037618 0.08040201 0.83333333 0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.        ]]
```

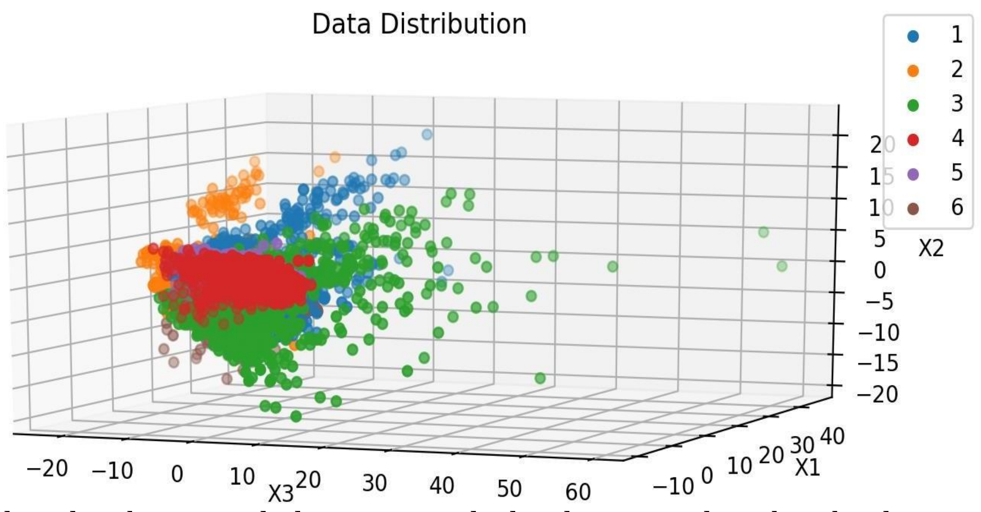
Part B – Human Activity Classification

Dataset Links –

https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones
https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

The given information states that there are 561 features, which is a substantial number, but it comes with some drawbacks, such as some features being redundant and not contributing to classification, and correlated features providing similar information. Additionally, it takes more computation time. Consequently, to mitigate these issues, PCA is employed as a technique for reducing dimensionality. It transforms the features into a smaller number of relevant ones by maximizing the variance.

Data Visualization after applying PCA-



From the data distribution and class priors calculated, it is evident that the data is evenly distributed between all the classes. The regularization parameter and assumptions regarding class-conditional PDF are similar to the previous part.

$$\mathbf{C} = \begin{matrix} \mathbf{0.912} & \mathbf{0.023} & \mathbf{0.029} & \mathbf{0} & \mathbf{0.0014} & \mathbf{0} \\ \mathbf{0.056} & \mathbf{0.919} & \mathbf{00.125} & \mathbf{0.0015} & \mathbf{0} & \mathbf{0} \\ \mathbf{0.021} & \mathbf{0.0567} & \mathbf{0.845} & \mathbf{0} & \mathbf{0} & \mathbf{0.0007} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0.604} & \mathbf{0.08} & \mathbf{0.032} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0.375} & \mathbf{0.915} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0.0398} & \mathbf{0.002} & \mathbf{0.967} \end{matrix}$$

Minimum Expected risk is 0.091.

The Wine Quality Classification problem is better solved using a Gaussian classifier, which produces more accurate results. This is because the dataset is balanced, meaning that the class priors are similar. As a result, the class posteriors depend largely on the class-conditional distributions, and the model can establish correlations between the reduced features and class labels. While principal component analysis (PCA) is a useful method for extracting significant information from the extensive feature set, it is important to evaluate the model on new samples to avoid overfitting to the training set.

APPENDIX

1A

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
np.random.seed(10)

def generate_class_labels(p):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[0]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    # generate samples for each class
    samples = np.zeros(shape = [sample_size, features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            samples[index, :] = np.random.multivariate_normal(mean_0, covariance_0)
        elif class_labels[index] == 1:
            samples[index, :] = np.random.multivariate_normal(mean_1, covariance_1)
    return samples

def discriminant_score(samples, mean_0, mean_1, covariance_0, covariance_1):
    gaussian_pdf_0 = np.log(multivariate_normal.pdf(samples, mean_0, covariance_0))
    gaussian_pdf_1 = np.log(multivariate_normal.pdf(samples, mean_1, covariance_1))
    return gaussian_pdf_1 - gaussian_pdf_0

def calculate_mid_points(d_score_sorted):
    threshold = []
    for i in range( len(d_score_sorted) - 1 ):
        threshold.append( (d_score_sorted[i] + d_score_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(d_score, threshold, p):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (d_score >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] =  false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[1] * false_positive[index]) + (p[0] * (1 - true_positive[index]))
    return true_positive, false_positive, error

# Number of samples
sample_size = 10000
features = 4
# class-conditional Gaussian pdf parameters
mean_0 = np.array([-1/2, -1/2, -1/2, -1/2])
covariance_0 = np.array([[2/4, -0.5/4, 0.3/4, 0], [-0.5/4, 1/4, -0.5/4, 0], [0.3/4, -0.5/4, 1/4, 0], [0, 0, 
0, 2/4]])
```

```
mean_1 = np.array([1, 1, 1, 1])
covariance_1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])
# class priors
p = [0.35, 0.65]

# generating class labels for the given number of samples
class_labels = generate_class_labels(p)

# generating the samples / data
samples = generate_samples(sample_size, class_labels)

#Plot Data / Samples
fig = plt.figure()
ax = plt.axes(projection = "3d")
samples_class0 = ax.scatter(samples[(class_labels==0),3], samples[(class_labels==0),1],
samples[(class_labels==0),2],'+',color ='blue', label="0")
samples_Class1 =
ax.scatter(samples[(class_labels==1),3],samples[class_labels==1,1],samples[class_labels==1,2],'.',c
olor = 'red', label="1")
plt.xlabel('X1')
plt.ylabel('X3')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()

# calculate discriminant score
d_score = discriminant_score(samples, mean_0, mean_1, covariance_0, covariance_1)

d_score_sorted = np.sort(d_score)
# calculating the threshold values (mid - points of d_score)
threshold = calculate_mid_points(d_score_sorted)

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(d_score, threshold, p)
# Plot ROC curve
plt.plot(false_positive, true_positive, color = 'yellow')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(false_positive[np.argmin(error)], true_positive[np.argmin(error)],'o',color = 'black')
plt.show()
print(f'Value of Gamma (practical) is {round(np.exp(threshold[np.argmin(error)]), 5)} ')
print(f'Corresponding minimum error is {round(np.min(error), 5)}')

# computing idea value
ideal_gamma = p[1] / p[0]
ideal_threshold = np.log(ideal_gamma)
ideal_decision = (d_score >= ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
1))))/np.size(np.where(class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
0))))/np.size(np.where(class_labels == 0))
ideal_error = (p[1] * ideal_false_positive) + (p[0] * (1 - ideal_true_positive))
print(f'Value of Gamma (Ideal) is {round(ideal_gamma, 5)} ')
print(f'Corresponding minimum error is {round(ideal_error, 5)}')
```

1B

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
np.random.seed(10)

def generate_class_labels(p):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[0]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    # generate samples for each class
    samples = np.zeros(shape = [sample_size, features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            samples[index, :] = np.random.multivariate_normal(mean_0, covariance_0)
        elif class_labels[index] == 1:
            samples[index, :] = np.random.multivariate_normal(mean_1, covariance_1)
    return samples

def discriminant_score(samples, mean_0, mean_1, features):
    gaussian_pdf_0 = np.log(multivariate_normal.pdf(samples, mean_0, np.eye(features, features)))
    gaussian_pdf_1 = np.log(multivariate_normal.pdf(samples, mean_1, np.eye(features, features)))
    return gaussian_pdf_1 - gaussian_pdf_0

def calculate_mid_points(d_score_sorted):
    threshold = []
    for i in range( len(d_score_sorted) - 1 ):
        threshold.append( (d_score_sorted[i] + d_score_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(d_score, threshold, p):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (d_score >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] =  false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[1] * false_positive[index]) + (p[0] * (1 - true_positive[index]))
    return true_positive, false_positive, error

# Number of samples
sample_size = 10000
features = 4
# class-conditional Gaussian pdf parameters
mean_0 = np.array([-1/2, -1/2, -1/2, -1/2])
```

```
covariance_0 = np.array([[2/4, -0.5/4, 0.3/4, 0], [-0.5/4, 1/4, -0.5/4, 0], [0.3/4, -0.5/4, 1/4, 0], [0, 0,
0, 2/4]])
mean_1 = np.array([1, 1, 1, 1])
covariance_1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])
# class priors
p = [0.35, 0.65]

# generating class labels for the given number of samples
class_labels = generate_class_labels(p)

# generating the samples / data
samples = generate_samples(sample_size, class_labels)

#Plot Data / Samples
fig = plt.figure()
ax = plt.axes(projection = "3d")
samples_class0 = ax.scatter(samples[(class_labels==0),3], samples[(class_labels==0),1],
samples[(class_labels==0),2],'+',color ='blue', label="0")
samples_Class1 =
ax.scatter(samples[(class_labels==1),3],samples[class_labels==1,1],samples[class_labels==1,2],'.',c
olor = 'red', label="1")
plt.xlabel('X1')
plt.ylabel('X3')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()

# calculate discriminant score
d_score = discriminant_score(samples, mean_0, mean_1, features)

d_score_sorted = np.sort(d_score)
# calculating the threshold values (mid - points of d_score)
threshold = calculate_mid_points(d_score_sorted)

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(d_score, threshold, p)
# Plot ROC curve
plt.plot(false_positive, true_positive, color = 'yellow')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(false_positive[np.argmin(error)], true_positive[np.argmin(error)],'o',color = 'black')
plt.show()
print(f'Value of Gamma (practical) is {round(np.exp(threshold[np.argmin(error)]), 5)} ')
print(f'Corresponding minimum error is {round(np.min(error), 5)}')

# computing idea value
ideal_gamma = p[1] / p[0]
ideal_threshold = np.log(ideal_gamma)
ideal_decision = (d_score >= ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
1))))/np.size(np.where(class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
0))))/np.size(np.where(class_labels == 0))
```

```
ideal_error = (p[1] * ideal_false_positive) + (p[0] * (1 - ideal_true_positive))
print(f'Value of Gamma (Ideal) is {round(ideal_gamma, 5)} ')
print(f'Corresponding minimum error is {round(ideal_error, 5)}')
```

1C

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from numpy import linalg as linA
np.random.seed(10)

def generate_class_labels(p):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[0]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    # generate samples for each class
    samples = np.zeros(shape = [sample_size, features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            samples[index, :] = np.random.multivariate_normal(mean_0, covariance_0)
        elif class_labels[index] == 1:
            samples[index, :] = np.random.multivariate_normal(mean_1, covariance_1)
    return samples

def calculate_mid_points(y_sorted):
    threshold = []
    for i in range( len(y_sorted) - 1 ):
        threshold.append( (y_sorted[i] + y_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(y, threshold, p):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (y >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] =  false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[1] * false_positive[index]) + (p[0] * (1 - true_positive[index]))
    return true_positive, false_positive, error

def between_slass_SB(mean_0, mean_1):
    return ( (mean_0 - mean_1) * (np.transpose(mean_0 - mean_1)) )

def within_class_SW(covariance_0, covariance_1):
    return ( covariance_0 + covariance_1 )
```

```
# Data projection using LDA
def data_projection(class0, class1, w_max):
    y0 = [0] * len(class0)
    y1 = [0] * len(class1)
    y0 = np.dot(np.transpose(w_max), np.transpose(class0))
    y1 = np.dot(np.transpose(w_max), np.transpose(class1))
    y =  np.concatenate( [y0, y1] )
    return y0, y1, y

# Number of samples
sample_size = 10000
features = 4
# class-conditional Gaussian pdf parameters
mean_0 = np.array([-1/2, -1/2, -1/2, -1/2])
covariance_0 = np.array([[2/4, -0.5/4, 0.3/4, 0], [-0.5/4, 1/4, -0.5/4, 0], [0.3/4, -0.5/4, 1/4, 0], [0, 0,
0, 2/4]])
mean_1 = np.array([1, 1, 1, 1])
covariance_1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])
# class priors
p = [0.35, 0.65]

#Compute scatter matrix within class
s_w = within_class_SW(covariance_0, covariance_1)

#Computer scatter matrix between class
s_b = between_slass_SB(mean_0, mean_1)

# generating class labels for the given number of samples
class_labels = generate_class_labels(p)

# generating the samples / data
samples = generate_samples(sample_size, class_labels)

#Computer Eigen Values and Eigen Vectors
v, w = linA.eig( linA.inv(s_w) * s_b )

# Maximum optimization objective
w_max = w[np.argmax(v)]
class0 = samples[ np.where(class_labels == 0) ]
class1 = samples[ np.where(class_labels == 1) ]

# Data projection using LDA
y0, y1, y = data_projection(class0, class1, w_max)

y_sorted = np.sort(y)

# calculating the threshold values (mid - points of d_score)
threshold = calculate_mid_points(y_sorted)

#Plot Data / Samples
fig = plt.figure()
ax = plt.axes(projection = "3d")
samples_class0 = ax.scatter(samples[(class_labels==0),3], samples[(class_labels==0),1],
samples[(class_labels==0),2],'+',color ='blue', label="0")
```

```
samples_Class1 =
ax.scatter(samples[(class_labels==1),3],samples[class_labels==1,1],samples[class_labels==1,2],'.',c
olor = 'red', label="1")
plt.xlabel('X1')
plt.ylabel('X3')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(y, threshold, p)

# Plot ROC curve
plt.plot(false_positive, true_positive, color = 'yellow')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(false_positive[np.argmin(error)], true_positive[np.argmin(error)],'o',color = 'black')
plt.show()
print(f'Value of Tau (practical) is {round(threshold[np.argmin(error)], 5)} ')
print(f'Corresponding minimum error is {round(np.min(error), 5)}')

# computing idea value
ideal_gamma = p[1] / p[0]
ideal_threshold = np.log(ideal_gamma)
ideal_decision = (y >= ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
1))))/np.size(np.where(class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
0))))/np.size(np.where(class_labels == 0))
ideal_error = (p[1] * ideal_false_positive) + (p[0] * (1 - ideal_true_positive))
print(f'Value of Tau (Ideal) is {round(ideal_gamma, 5)} ')
print(f'Corresponding minimum error is {round(ideal_error, 5)}')
```

2

```
import random
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D
np.set_printoptions(threshold=np.inf)

def generate_class_labels(p, sample_size):
    cummulative_sum = np.cumsum(p)
    random_labels = np.random.rand(sample_size)
    class_label = np.zeros([sample_size])
    for i in range(0, sample_size-1):
        if random_labels[i] <= cummulative_sum[0]:
            class_label[i] = 0
        elif random_labels[i] <= cummulative_sum[1]:
            class_label[i] = 1
        else:
            class_label[i] = 2
    return class_label

# Generate gaussian distribution for 10000 samples using mean and covariance matrices for each
label
def generate_samples(sample_size, features, mean, covariance_matrix):
    samples = np.zeros(shape = [sample_size, features])
    for i in range(sample_size):
        if (class_label[i] == 0):
            samples[i, :] = np.random.multivariate_normal(mean[0, :], covariance_matrix[0, :, :])
        elif (class_label[i] == 1):
            samples[i, :] = np.random.multivariate_normal(mean[1, :], covariance_matrix[1, :, :])
        elif (class_label[i] == 2):
        # Split samples based on mixture weights
            if (np.random.rand(1,1) >= weights[1]):
                samples[i, :] = np.random.multivariate_normal(mean[2, :], covariance_matrix[2, :, :])
            else:
                samples[i, :] = np.random.multivariate_normal(mean[3, :], covariance_matrix[3, :, :])

    return samples

def compute_class_conditional_pdf(class_labels, sample_size, mean, covariance_matrix):
    P_x_given_L = np.zeros(shape = [class_labels, sample_size])
    for i in range(class_labels):
        P_x_given_L[i, :] = multivariate_normal.pdf(samples, mean[i, :], covariance_matrix[i, :,:])

    return P_x_given_L

# Compute Class Posteriors using priors and class conditional PDF
def compute_classPosteriors(p, P_x_given_L, sample_size, class_labels):
    P_x = np.matmul(p, P_x_given_L)
    cp = (P_x_given_L * (np.matlib.repmat(np.transpose(p), 1, sample_size))) /
np.matlib.repmat(P_x, class_labels, 1)
    return cp


def cofusion_matrix(class_labels, Decision, class_label):
    cm = np.zeros(shape = [class_labels, class_labels])
    for d in range(class_labels):
        for l in range(class_labels):
            cm[d, l] = (np.size(np.where((d == Decision) & (l == class_label)))) /
np.size(np.where(class_label==l))
    return cm

sample_size = 10000              # Number of Samples
features = 3          # Number of features
class_labels = 3           # Number of classes
mixtures = 4          # Number of gaussian distributions

#Seed to obtain same results for random numbers
np.random.seed(42)

# Class Priors
p = np.array([[0.3, 0.3, 0.4]])
```

```
# Mean vectors
mean = np.zeros(shape=[mixtures, features])
mean[0, :] = [0, 0, 15]
mean[1, :] = [0, 15, 0]
mean[2, :] = [15, 0, 0]
mean[3, :] = [15, 0, 15]

# Covariance matrices
covariance_matrix = np.zeros(shape=[mixtures, features, features])
covariance_matrix[0, :, :] = 36 * np.linalg.matrix_power((np.eye(features)) + (0.01 *
np.random.randn(features, features)), 2)
covariance_matrix[1, :, :] = 36 * np.linalg.matrix_power((np.eye(features)) + (0.02 *
np.random.randn(features, features)), 2)
covariance_matrix[2, :, :] = 36 * np.linalg.matrix_power((np.eye(features)) + (0.03 *
np.random.randn(features, features)), 2)
covariance_matrix[3, :, :] = 36 * np.linalg.matrix_power((np.eye(features)) + (0.04 *
np.random.randn(features, features)), 2)

# Prior weights for gaussian components of class 2 and assigning labels
weights = [0.5,0.5]

class_label = generate_class_labels(p, sample_size)
samples = generate_samples(sample_size, features, mean, covariance_matrix)

#Select appropriate loss matrix and comment other two
loss_matrix_A = np.ones(shape = [class_labels, class_labels]) - np.eye(class_labels)
loss_matrix_B = np.array([[0, 1, 10], [1, 0, 10], [1, 1, 0]])
loss_matrix_C = np.array([[0, 1, 100], [1, 0, 100], [1, 1, 0]])

print("1. LOSS MATRIX A : ")
print(loss_matrix_A)
print("2. LOSS MATRIX B : ")
print(loss_matrix_B)
print("3. LOSS MATRIX C : ")
print(loss_matrix_C)

choice = int(input("\nSelect 1 or 2 or 3 : "))
if (choice == 1):
    loss_matrix = loss_matrix_A
elif (choice == 2):
    loss_matrix = loss_matrix_B
elif (choice == 3):
    loss_matrix = loss_matrix_C

print(f'\nThe loss matrix used is : \n{loss_matrix} \n')

# Compute Class conditional PDF
P_x_given_L = compute_class_conditional_pdf(class_labels, sample_size, mean,
covariance_matrix)

#compute classposteriors
cp = compute_classPosteriors(p, P_x_given_L, sample_size, class_labels)

# Evaluate Expected risk and decisions based on minimum risk
ExpectedRisk = np.matmul(loss_matrix, cp)
```

```
Decision = np.argmin(ExpectedRisk, axis = 0)
avg_exp_risk = np.sum(np.min(ExpectedRisk, axis = 0)) / sample_size
print(f'The Average Expected Risk is : {avg_exp_risk} \n')

# Estimate Confusion Matrix
cm = cofusion_matrix(class_labels, Decision, class_label)
print(f'Confusion Matrix : \n {cm}')

# Plot Classification results
fig = plt.figure()
ax = plt.axes(projection = "3d")
ax.scatter(samples[(class_label==2) & (Decision == 1),0], samples[(class_label==2) & (Decision
== 1),1], samples[(class_label==2) & (Decision == 1),2],color ='red', marker = 's')
ax.scatter(samples[(class_label==2) & (Decision == 2),0], samples[(class_label==2) & (Decision
== 2),1], samples[(class_label==2) & (Decision == 2),2],color ='green', marker = 's')
ax.scatter(samples[(class_label==2) & (Decision == 0),0], samples[(class_label==2) & (Decision
== 0),1], samples[(class_label==2) & (Decision == 0),2],color ='red', marker = 's')
ax.scatter(samples[(class_label==1) & (Decision == 1),0], samples[(class_label==1) & (Decision
== 1),1], samples[(class_label==1) & (Decision == 1),2],color ='green', marker = '^')
ax.scatter(samples[(class_label==1) & (Decision == 2),0], samples[(class_label==1) & (Decision
== 2),1], samples[(class_label==1) & (Decision == 2),2],color ='red', marker = '^')
ax.scatter(samples[(class_label==1) & (Decision == 0),0], samples[(class_label==1) & (Decision
== 0),1], samples[(class_label==1) & (Decision == 0),2],color ='red', marker = '^')
ax.scatter(samples[(class_label==0) & (Decision == 0),0], samples[(class_label==0) & (Decision
== 0),1], samples[(class_label==0) & (Decision == 0),2],color ='green', marker = '*')
ax.scatter(samples[(class_label==0) & (Decision == 2),0], samples[(class_label==0) & (Decision
== 2),1], samples[(class_label==0) & (Decision == 2),2],color ='red', marker = '*')
ax.scatter(samples[(class_label==0) & (Decision == 1),0], samples[(class_label==0) & (Decision
== 1),1], samples[(class_label==0) & (Decision == 1),2],color ='red', marker = '*')
plt.xlabel('X1')
plt.ylabel('X2')
ax.set_zlabel('X3')
plt.title('Classification Plot')
plt.show()
```

3A

```
import random
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from numpy import linalg as LA

# compute mean and covariance
def compute_mean_and_covariance(No_samples, No_features, class_labels, No_labels):
    unq_ls = len(np.unique(class_labels))
    mean = np.zeros( [No_labels, No_features] )
    covariance = np.zeros( [No_labels, No_features, No_features] )

#     for i, cls_i in enumerate(np.unique(class_labels)):
#         mean[i, :] = np.mean(df[df['quality'] == cls_i].drop(columns=['quality']), axis = 0)
#        covariance[i, :, :] = np.cov(df[(df['quality'] == cls_i)].drop(columns=['quality']), rowvar =
False)
#        covariance[i, :, :] += (0.00000005) * ((np.trace(covariance[i, :, :])) /
LA.matrix_rank(covariance[i, :, :])) * np.eye(No_features)
    for i in range(No_labels):
        mean[i, :] = np.mean(df[df['quality'] == i].drop(columns=['quality']), axis = 0)

        if (i not in class_labels):
            covariance[i, :, :] = np.eye(No_features)
        else:
            covariance[i, :, :] = np.cov(df[df['quality'] == i].drop(columns=['quality']), rowvar = False)
            covariance[i, :, :] += (0.00000005) * ((np.trace(covariance[i, :,
:]))/LA.matrix_rank(covariance[i, :, :])) * np.eye(No_features)

    return mean, covariance

# compute class condtional pdf
def compute_class_conditional_pdf(class_labels, No_labels, No_samples, mean,
covariance_matrix):
    P_x_given_L = np.zeros(shape = [No_labels, No_samples])
    unq_ls = np.unique(class_labels)
    for i in unq_ls:
        P_x_given_L[i, :] = multivariate_normal.pdf(df.drop(columns = ['quality']), mean[i, :],
covariance_matrix[i, :,:])
    return P_x_given_L

# compute class priors based on sample count
def class_priors(No_labels, class_labels, No_samples):
    priors = np.zeros(shape = [11, 1])
    for i in range(0, No_labels):
        priors[i] = (np.size(class_labels[np.where((class_labels == i))])) / No_samples
    return priors

# compute Confusion Matrix
def compute_confusion_matrix (No_labels, class_labels):
    cm = np.zeros(shape = [No_labels, No_labels])
    for i in range(No_labels):
        for j in range(No_labels):
            if j in class_labels and i in class_labels:
                cm[i, j] = (np.size(np.where((i == Decision) & (j == class_labels)))) /
np.size(np.where(class_labels == j))
    return cm

# Import Dataset
df = pd.read_csv(r'C:\Users\Shiva Kumar Dande\Downloads\winequality-red.csv')
df = df.dropna()
Data = df.to_numpy()
#print(Data)
No_Classes = 11

rows, columns = np.shape(df)
No_samples = rows
No_features = columns - 1
No_labels = 11
df.dropna(inplace=True)
```

```
  class_labels = df['quality']
  class_labels = np.array(class_labels)

  # Plot Data Distribution
  fig = plt.figure()
  ax = plt.axes(projection = "3d")
  for i in range(No_labels):
     ax.scatter(Data[(class_labels==i),1],Data[(class_labels==i),4],Data[(class_labels==i),8], label =
  i)
  plt.xlabel('X3')
  plt.ylabel('X1')
  ax.set_zlabel('X2')
  ax.legend()
  plt.title('Data Distribution')
  plt.show()

  loss_matrix = np.ones([No_labels, No_labels]) - np.eye(No_labels)
  mean, covariance = compute_mean_and_covariance(No_samples, No_features, class_labels,
  No_labels)
  pdf = compute_class_conditional_pdf(class_labels, No_labels, No_samples, mean, covariance)
  p = class_priors(No_labels, class_labels, No_samples)

  # Compute Class Posteriors using priors and class conditional PDF
  P_x = np.matmul(np.transpose(p), pdf)
  class_posteriors = (pdf * (np.matlib.repmat(p, 1, No_samples))) / np.matlib.repmat(P_x, No_labels,
  1)

  # Evaluate Expected risk and decisions based on minimum risk
  expected_risk = np.matmul(loss_matrix, class_posteriors)
  Decision = np.argmin(expected_risk, axis = 0)
  avg_exp_risk = np.sum(np.min(expected_risk, axis = 0)) / No_samples
  print(f'The Average Expected Risk is {avg_exp_risk}')
  cm = compute_confusion_matrix (No_labels, class_labels)
  print(cm)
```

3B

```
import pandas as pd
import random
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D
from numpy import linalg as LA
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler



def compute_mean_and_covariance(No_samples, No_features, class_labels, No_labels):
    mean = np.zeros(shape = [No_labels, No_features])
    covariance = np.zeros(shape = [No_labels, No_features, No_features])
    for i in range(0, No_labels):
        mean[i, :] = np.mean(Data[(class_labels == i + 1), :], axis = 0)
        covariance[i, :, :] = np.cov(Data[(class_labels == i + 1), :], rowvar = False)
        covariance[i, :, :] += (0.00001) * ((np.trace(covariance[i,:,:]))/LA.matrix_rank(covariance[i,:,:]))
* np.eye(10)
        #Check if covariance matrices are ill-conditioned
        #print(LA.cond(covariance_matrix[i,:,:]))
    return mean, covariance

def compute_class_conditional_pdf(class_labels, No_labels, No_samples, mean, covariance_matrix):
    P_x_L = np.zeros(shape = [No_labels, No_samples])
    for i in range(0, No_labels):
        P_x_L[i, :] = multivariate_normal.pdf(Data, mean[i, :], covariance[i, :,:])
    return P_x_L

def class_priors(No_labels, class_labels, No_samples):
    p = np.zeros(shape = [No_labels, 1])
    for i in range(0, No_labels):
        p[i] = (np.size(label[np.where((class_labels == i + 1))])) / No_samples
    return p

def compute_confusion_matrix (No_labels, class_labels):
    cm = np.zeros(shape = [No_labels, No_labels])
    for d in range(No_labels):
        for l in range(No_labels):
            cm[d, l] = (np.size(np.where((d == Decision) & (l == class_labels - 1)))) /
np.size(np.where(class_labels - 1 == l))
    return cm


df = pd.read_csv(r'C:\Users\Shiva Kumar Dande\Desktop\Machine Learning Projects\Assignment
1\HumanDetection\X_train.txt')
Data = df.to_numpy()

No_samples = Data.shape[0]
Y = pd.read_csv(r'C:\Users\Shiva Kumar Dande\Desktop\Machine Learning Projects\Assignment
1\HumanDetection\y_train.txt')
class_labels = np.squeeze(Y.to_numpy())

Data = Data[:, 0:-2]
sc = StandardScaler()
Data = sc.fit_transform(Data)

pca = PCA(n_components = 10)
Data = pca.fit_transform(Data)

No_labels = 6           # Number of labels
No_features = 10         # Number of features

# Plot Data Distribution
fig = plt.figure()
ax = plt.axes(projection = "3d")
for i in range(1, N_labels + 1):
    ax.scatter(Data[(class_labels==i),1],Data[(class_labels==i),2],Data[(class_labels==i),3],
class_labels=i)
```

```
plt.xlabel('X3')
plt.ylabel('X1')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()

loss_matrix = np.ones(shape = [No_labels, No_labels]) - np.eye(No_labels)
mean, covariance = compute_mean_and_covariance(No_samples, No_features, class_labels,
No_labels)
pdf = compute_class_conditional_pdf(class_labels, No_labels, No_samples, mean, covariance)
p = class_priors(No_labels, class_labels, No_samples)

P_x = np.matmul(np.transpose(priors), P_x_given_L)
ClassPosteriors = (P_x_given_L * (np.matlib.repmat(priors, 1, N))) / np.matlib.repmat(P_x, N_labels,
1)
#expected risk
er = np.matmul(loss_matrix, ClassPosteriors)
Decision = np.argmin(er, axis = 0)

print("Average Expected Risk", np.sum(np.min(er, axis = 0)) / No_samples)
cm = compute_confusion_matrix (No_labels, class_labels)
np.set_printoptions(suppress=True)
print(cm)
```